SwinxsCoreTalk

Documentation

Copyright © 2009 Swinxs BV
The contents of this document is confidential and protected by Dutch law and international treaties. Nothing from this document may be disclosed or copied, by any means, electronic, magnetic or by photocopy, without the prior written permission of the publisher.

Swinxs BV
Paasheuvelweg 29
1105 BG Amsterdam
The Netherlands

mail: info@swinxs.com
http://www.swinxs.com

KvK Rivierenland: 17118466
BTW: NL 808347433B01
Rek. nr.: Rabobank 13.06.47.268
IBAN: NL89 RABO 0130 6472 68
Swift Code BIC: RABONL2U

Release history

Version	*Date*	*Author*	*Comments*
1.0	2009-01-01	Mark Ruys	Initial version, migrated from the SwinxsSDK documentation.
1.0a	2009-07-31	Mark Ruys	Dedicated timer 13 for SwinxsTalk
1.1	2009-10-02	Mark Ruys	Added DELETE boolean for deleting samples

Table of contents

1. Summary....4
2. Introduction....5
  2.1 Swinxs Source....5
  2.2 SwinxsEmulator....5
3. System description....6
  3.1 Sensors, inputs and outputs....6
  3.2 SwinxsCoreTalk Virtual machine....8
4. SwinxsCoreTalk....10
  4.1 Interface....10
  4.2 Instructions....14
  4.3 Example....17
5. SwinxsCoreTalk subsystems....20
  5.1 File system....20
  5.2 RFID tags....22
  5.3 Audio....22
  5.4 Move & shake & noise....24
  5.5 Buttons....24
  5.6 Pulse....25
  5.7 Menu....26
  5.8 Timers....29
  5.9 Shuffle....29
  5.10 Game exit....30
  5.11 Power management....30
6. SwinxsEmulator....31
  6.1 SwinxsCoreTalk Preprocessor....32
  6.2 SwinxsCoreTalk Debugger....33
  6.3 Tips....33

1. Summary

This document describes SwinxsCoreTalk. It should help programmers to maintain and extend the SwinxsSDK. For writing Swinxs games, the higher level language SwinxsTalk should be used. See the SwinxsSDK documentation for more details.
You can download the latest version of the SwinxsSDK at www.swinxs.com/download. For support, latest versions, etc, see the developers community at www.swinxs.com/games.

Intended audience

Target for this documentation are SwinxsSDK developers. It is assumed that the reader has a general (BSc level) technical background. Also, the reader is assumed to be familiar with the SwinxsSDK documentation.

2. Introduction

Swinxs is a high-tech games console, sustainably designed for active games you can play inside or outdoors. The unique XS-wristband gives you access to a new world to play games, listen to music, experience exciting adventures and make fantasies reality. Connect your Swinxs to your computer and on your own Swinxs page you can, as well as many other things, keep track of your scores and download new games.
Swinxs implements a virtual machine running SwinxsCoreTalk programs.

2.1 Swinxs Source

The Swinxs Source itself is divided in audio material and the source code of games. This source code is written in a dedicated language SwinxsCoreTalk or in a higher level language called SwinxsTalk. The audio material consist of uncompressed WAV files. You can find the audio samples on your Swinxs.

SwinxsCoreTalk

SwinxsCoreTalk is a low level language to specify Swinxs games. It's concepts are related to PLC's (Programmable Logic Controllers). A PLC basically maps inputs (XS-wristbands read, buttons, accelerometer) to outputs (LEDs, audio).
The SwinxsCoreTalk games can be executed by the virtual machine (VM) running the Swinxs.

SwinxsTalk

Because it is rather tedious to develop games in SwinxsCoreTalk, we developed a higher-level language called SwinxsTalk. We expect that anyone with some basic programming skills should be able to get started quickly and develop his or her own games. The syntax of SwinxsTalk is slightly similar to languages like JavaScript.

2.2 SwinxsEmulator

The SwinxsEmulator has two different purposes. Firstly it enables you to develop and debug your Swinxs games. Secondly, it can compile all your games to a binary format which a Swinxs can execute. It will publish your games to your Swinxs, or upload it for you to the swinxs.com site.

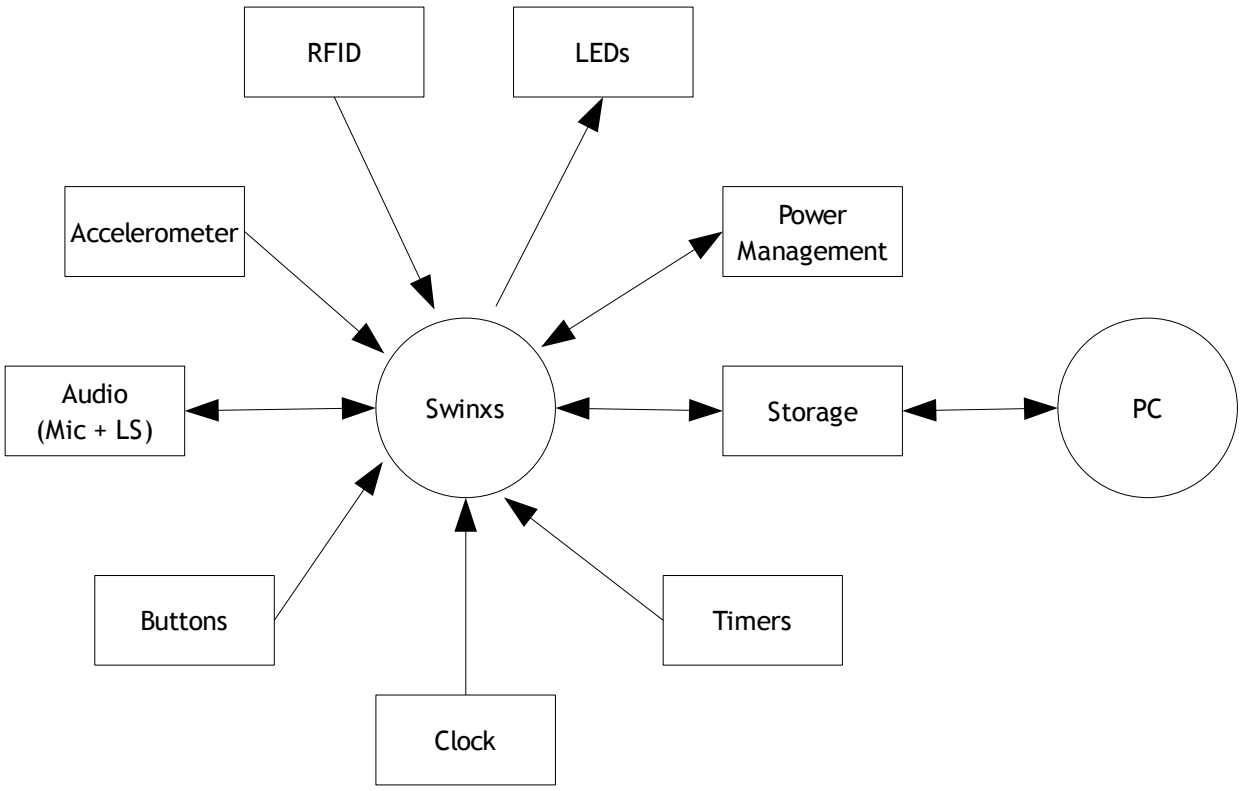
3. System description

Swinxs is an open gaming platform that acts as a catalyst for social physical game play for 3 to 12 year old children. Swinxs guides children through a set of games.

The Swinxs hardware is a simple device with only three buttons, however more hidden sensors are available. The area enclosed by the three buttons is capable of lighting up over 60 different colors. But the main interfaces of Swinxs are its RFID sensor and sound output device.
Swinxs supports multiple languages.

3.1 Sensors, inputs and outputs

Swinxs has the following in- and outputs available:



RFID

RFID tags can be detected and read. Each tag has a unique 24-bit number (ID), plus 16 bits to mark special features. Currently we distinguish between XS-tags (bracelets) and playing cards.

Buttons

Swinxs has three buttons (white, green and red). Each button has 4 triggers associated with it:

- button is being hold down
- button has been pressed shortly (less the 500 msec)
- button is pressed long (longer than 500 msec)
- button is pressed and hold down for more than 2 seconds

LEDs

Swinxs has 3 LEDs blended to a single color. Each LED can be set to 32 intensities, ranging from blacked out to bright white.

Accelerometer

You receive events when Swinxs is moved or when Swinxs is shaken. The thresholds which define the sensitivity can be modified through two settings: length and intensity.

Loudspeaker

The audio system is able to play uncompressed wave files. You can play an audio sample, and abort it if you want so. You are notified while audio is being played. During playback, you can adjust the volume. Swinxs can mix audio streams real time.
Typically, Swinxs audio files are uncompressed wave-files sampled at 22 Khz, 16 bit depth, mono.

Microphone

Microphone to detect someone is talking (loud). It's just a trigger.
Also, Swinxs is able to perform voice recordings which are saved to Swinxs' storage.

Timers

Swinxs has 13 freely available timers running at a resolution of 100 msec. Each timer can be initialized, started, stopped and read out. Maximum value is 65535, which holds roughly 100 minutes.

Clock

As long as the Swinxs battery does not run out of power, Swinxs maintains a real time clock.

Power control

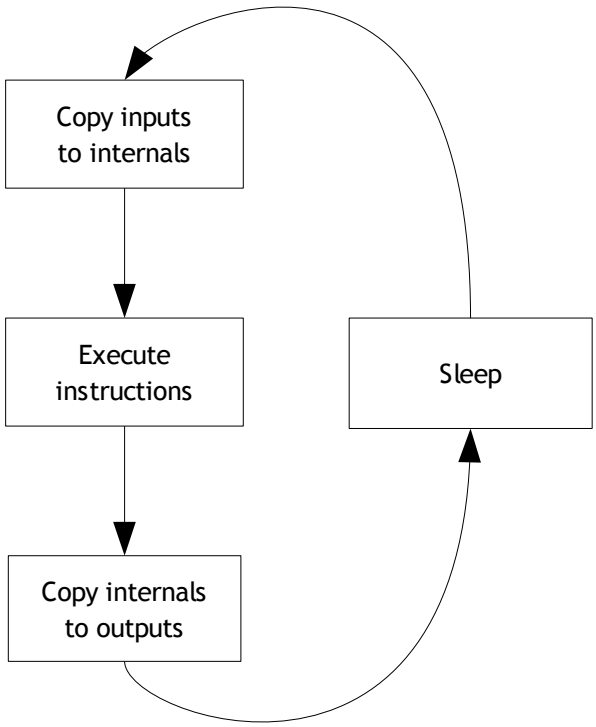
When the Swinxs reaches the low power condition, it is detected as an input. You can power off Swinxs from within your application.

Storage

Integer application variables can be saved to persistent memory and loaded back into Swinxs memory on a per-game basis.
When Swinxs is attached to a PC, the persistent memory is available as a USB flash drive. This drive contains all applications (games), game meta-information, audio samples, scores, saved Swinxs memory, etc.

3.2 SwinxsCoreTalk Virtual machine

Swinxs games are coded in a specially designed language called SwinxsCoreTalk. SwinxsCoreTalk code is interpreted by a virtual machine. This virtual machine is implemented as an finite state machine.



Control cycle

The virtual machine basically reads all inputs, process a set of instructions (the game definition) and write all outputs. Completing a full cycle of the above finite state machine is referred to as a control cycle. The sleep delay is typical 40 msec but may fluctuate[1]. When Swinxs is busy, control cycles may even be dropped.

Accumulator and register

The core of the virtual machine consist of binary accumulator and a 16-bit word register. A word is a unsigned integer from 0 up to 65535. You could regard the accumulator and register as the main memory of a Swinxs.

Variables

The virtual machine further supports 256 binary variables, some of which are virtual IO's, usually connected to real IO's and 256 word variables, some of which have a special purpose.
To avoid spurious signals on the outputs these are set after completion of the control cycle. For similar reasons the inputs are copied to internal variables before the program executes.

Instruction memory

Part of the virtual machine is also a 6000 byte instruction memory for game specifications. A single SwinxsCoreTalk instruction typically takes two bytes.

[1]This may change in the future. Use timers when you need an accurate notion of time.

4. SwinxsCoreTalk

This chapter describes the SwinxsCoreTalk language. First we describe the Swinxs low-level interface to the outside word. Next we describe all available instructions. We conclude with a very basic example.

4.1 Interface

The Swinxs virtual machine interfaces to the outside world via variables. The following tables list which variables are available and gives a very short description. Some variables are implemented only from a specific Swinxs firmware version. This is marked in the column FW. The column Type describes what kind of variable it is:

- R(ead), W(rite) or RW (both)
- In(put), Out(put)
- App means that the variable is handled in the SwinxsCoreTalk libraries and not part of the Swinxs firmware

Word variables

Swinxs possess 256 word variables of which the first 100 have predefined meanings or are reserved for future use. The others (100 up to 255) are free to use. Every word variable can contain a 16 bits unsigned value, so it ranges from 0 to 65535.

Mnemonic	*Type*	*FW*	*Description*
TIMER_0	RW		Timer with a resolution of 100 msec
TIMER_1	RW		Timer with a resolution of 100 msec
TIMER_2	RW		Timer with a resolution of 100 msec
TIMER_3	RW		Timer with a resolution of 100 msec
TIMER_4	RW		Timer with a resolution of 100 msec
TIMER_5	RW		Timer with a resolution of 100 msec
TIMER_6	RW		Timer with a resolution of 100 msec
TIMER_7	RW		Timer with a resolution of 100 msec
TIMER_8	RW		Timer with a resolution of 100 msec
TIMER_9	RW		Timer with a resolution of 100 msec
TIMER_10	RW		Timer with a resolution of 100 msec
TIMER_11	RW		Timer with a resolution of 100 msec
TIMER_12	RW		Timer with a resolution of 100 msec
TIMER_ST	RW		Dedicated timer with a resolution of 100 msec
TIMER_IDLE	RW		Dedicated timer with a resolution of 100 msec
TIMER_OPTIONS	RW		Dedicated timer with a resolution of 100 msec

Mnemonic	*Type*	*FW*	*Description*
RESIDUE	R		Temporary variable to hold the arithmetic overflow or remainder
EXCEPT_ID	R		In case of an exception, holds exception id
GAME	RW		Determines from which game directory te load files
SND_TYPE	W		Specifies sound type to play
SND_FILE	W		Specifies sound file to play

Mnemonic	*Type*	*FW*	*Description*
SND_VOLUME	W		Master sound volume from 0 (silence) to 100 (max)
SND_TYPE_BG	W	310	Specifies background sound type to play
SND_FILE_BG	W	310	Specifies background sound file to play
SND_VOLUME_BG	W	310	Background sound volume from 0 (silence) to 100 (max), relative to SND_VOLUME.
SND_VOLUME_B	App	310	Copied to SND_VOLUME_BG while no foreground is playing
SND_VOLUME_X	App	310	Copied to SND_VOLUME_BG while foreground is playing (mixed)
LED_COLOR	W		Color of the LEDs, modulo 64
SHUFFLE	RW		Shuffle value
MENU_TYPE	W		Menu type (language, audio or games)
MENU_SIZE	R		Number of items in the menu
MINPLAY	R		Property of selected game
MAXPLAY	R		Property of selected game
GAMETYPE	R		Property of selected game
AGECLASS	R		Property of selected game
SHAKE_TH	W		Shake threshold setting
SHAKE_CNT	W		Shake duration setting
MOVE_TH	W		Move threshold setting
MOVE_CNT	W		Move duration setting
NOISE_TH	W		Noise threshold setting

Mnemonic	*Type*	*FW*	*Description*
NOF_XS	R		Number of registered XSes
RFID_HI	R		RFID last tag detected divided by 65530
RFID_LO	R		RFID last tag detected modulo 65530
TAG_BITS	R		Special bits last tag detected
DATE	R	499	(((Year - 2008) * 12 + (Month - 1)) * 32 + Day) * 7 + DayOfWeek, where Sunday is 0, Monday is 1, etc.
TIME	R	310	Hours * 60 + Minutes
SCORE_0	W		Score XS 0 to be saved to disk
SCORE_1	W		Score XS 1 to be saved to disk
SCORE_2	W		Score XS 2 to be saved to disk
SCORE_3	W		Score XS 3 to be saved to disk
SCORE_4	W		Score XS 4 to be saved to disk
SCORE_5	W		Score XS 5 to be saved to disk
SCORE_6	W		Score XS 6 to be saved to disk
SCORE_7	W		Score XS 7 to be saved to disk
SCORE_8	W		Score XS 8 to be saved to disk
SCORE_9	W		Score XS 9 to be saved to disk

Mnemonic	*Type*	*FW*	*Description*
WORDVAR_0			Free for application use
WORDVAR_1			Free for application use
WORDVAR_2			Free for application use
WORDVAR_3			Free for application use
WORDVAR_4			Free for application use
WORDVAR_5			Free for application use
WORDVAR_6			Free for application use
WORDVAR_7			Free for application use
WORDVAR_8			Free for application use
WORDVAR_9			Free for application use

The following variables are not implemented in the VM, but in special SwinxsCoreTalk include files.

Mnemonic	*Type*	*FW*	*Description*
STATE	App		Used to implement the state machine.
LAST_XS	RW		Remember the last tag seen.
PREV_SND_TYPE	App		For internal use only.
PREV_SND_FILE	App		For internal use only.
PREV_GAME	App		For internal use only.
PULSE_PHASE	App		The phase the pulse is in (0 for stopped)
PULSE_COLOR_0	App		Defines the color of a pulse (phase 1 and 6)
PULSE_COLOR_1	App		Defines the color of a pulse (phase 2 and 5)
PULSE_COLOR_2	App		Defines the color of a pulse (phase 3 and 4)
PULSE_PACE	App		The speed of the pulse: roughly in units of 250 msec.
PULSE_TICK	App		For internal use only.
TIME_TO_SLEEP	App		The time of inactivity after which the Swinxs falls asleep (in 10ths of seconds).

Boolean variables

Swinxs possess 256 word variables of which the first 100 have predefined meanings or are reserved for future use. The others (100 up to 255) are free to use.

Mnemonic	*Type*	*FW*	*Description*
TIMCTRL_0	RW		Timer status, controls timer 0
TIMCTRL_1	RW		Timer status, controls timer 1
TIMCTRL_2	RW		Timer status, controls timer 2
TIMCTRL_3	RW		Timer status, controls timer 3
TIMCTRL_4	RW		Timer status, controls timer 4
TIMCTRL_5	RW		Timer status, controls timer 5
TIMCTRL_6	RW		Timer status, controls timer 6
TIMCTRL_7	RW		Timer status, controls timer 7
TIMCTRL_8	RW		Timer status, controls timer 8
TIMCTRL_9	RW		Timer status, controls timer 9
TIMCTRL_10	RW		Timer status, controls timer 10
TIMCTRL_11	RW		Timer status, controls timer 11
TIMCTRL_12	RW		Timer status, controls timer 12
TIMCTRL_ST	RW		Timer status, controls dedicated timer 13
TIMCTRL_IDLE	RW		Timer status, controls dedicated timer 14
TIMCTRL_OPTIONS	RW		Timer status, controls dedicated timer 15

Mnemonic	*Type*	*FW*	*Description*
WHITE_DOWN	In		True while the white button is pressed
WHITE_SHORT	In		Set when the white button is pressed shortly
WHITE_LONG	In		True while the white button is pressed long
WHITE_HOLD	In		True while the white button is pressed for more than 2 seconds
GREEN_DOWN	In		True while the green button is pressed
GREEN_SHORT	In		Set when the green button is pressed shortly
GREEN_PULSE	In		Set when the green button is pressed
GREEN_LONG	In		True while the green button is pressed long
GREEN_HOLD	In		True while the green button is pressed for more than 2 seconds

Mnemonic	*Type*	*FW*	*Description*
RED_DOWN	In		Set while the red button is pressed
RED_SHORT	In		Set when the red button is pressed shortly
RED_LONG	In		True while the red button is pressed long
RED_HOLD	In		True while the red button is pressed for more than 8 seconds

Mnemonic	*Type*	*FW*	*Description*
EXCEPTION	In		Set when an exception occurs
RESET	In		Set when a program is loaded and executed
HWRESET	In		Set when Swinxs is powered on[2]
WDRESET	In		Set when Swinxs is recovered from a crash
HAS_LANG	In		Set when the player has selected a language
MOVE	In		True while the Swinxs is moved
SHAKE	In		True while the Swinxs is shaken
PLAYING	In		True while playing audio
PLAYING_BG	In	310	True while playing background audio
NOISE	In		True while the Swinxs hears noise
RECORD	Out	100	If set, Swinxs makes a voice recording
OVERWRITE	Out	100	If set, voice recordings overwrite existing samples
DELETE	Out	400	If set, delete the samples specified by SND_FILE and SND_TYPE
SND_EXISTS	In	400	True if the file specified by SND_FILE and SND_TYPE exists
PLAY	Out		Rising edge starts playing audio; when reset playing will be aborted
PLAY_BG	Out	310	Rising edge starts playing background audio; when reset playing will be aborted
PAUSE	Out	500	If set, pause foreground audio
PAUSE_BG	Out	500	If set, pause background audio
LED_ENA	Out		If set, LEDs are enabled, otherwise blanked out
SHUF_ENA	Out		Should be set while using the shuffle subsystem
SHUF_READ	Out		Set when you get the next entry in the shuffle table
SHUF_REWIND	Out		Rewind the shuffle pointer to the beginning of the list
SHUF_DONE	In		True when the end of the shuffle list is reached
MENU_ENA	Out		Rising edge initializes the menu subsystem and fills the menu list
MENU_READ	Out		Read the next entry in the menu list
MENU_ACCEPT	Out		Accept the current item in the menu list
REVERSE	Out		If set, MENU_READ will give the previous entry in the list
BAT_LOW	In		True while the battery is on low power
SUICIDE	Out		If set, the Swinxs will be turned off

Mnemonic	*Type*	*FW*	*Description*
RF_ENA	Out		Reset to turn of RF detection in order to save some power
TAG	In		Set when a tag is detected
REGISTER	Out		A rising edge starts a new registration procedure
WRITE_SCORE	Out		Save scores of registered players to disk

Mnemonic	*Type*	*FW*	*Description*
BOOLVAR_0			Free for application use
BOOLVAR_1			Free for application use
BOOLVAR_2			Free for application use
BOOLVAR_3			Free for application use

[2]Supported from Swinxs firmware 100 and higher.

Mnemonic	*Type*	*FW*	*Description*
BOOLVAR_4			Free for application use
BOOLVAR_5			Free for application use
BOOLVAR_6			Free for application use
BOOLVAR_7			Free for application use
BOOLVAR_8			Free for application use
BOOLVAR_9			Free for application use

The following variables are not implemented in the VM, but in special SwinxsCoreTalk include files.

Mnemonic	*Type*	*FW*	*Description*
RETURN_VISIT	App		
XS	In		A registered XS is detected
IGNORE_BAT_LOW	App		Ignore a battery low condition
DONE	App		Done processing instructions in this control cycle
INITIALIZED	App		Set while Swinxs is initializing after a power on
GREEN_PULSE	App		The green button is pressed
GREEN	App		Do not use this variable
PLAY_VOLUME	App		Do not use this variable
PREV_PLAY	App		Do not use this variable
INITIALIZED	App		True when the Swinxs is initialized
PLAY_VISUAL	App		True if we want to visualize playing. Also, when false, Swinxs won't fall asleep.
ACCESSORY	App		True if a accessory tag is detected
GREEN	App		Do not use this variable

4.2 Instructions

Each instruction consists of a opcode and optionally an operand to operate on. All instructions are executed every control cycle. However, most instructions are conditionally in the sense that they are ignored when the accu false (or true).
A word is a 16 bit unsigned value which can hold an integer from 0 to 65535. A boolean is a one bit value which can hold true or false.

The following symbols are used:

Symbol	*Description*
A	The boolean accumulator (accu)
R	The word register
B	The array with 256 boolean variables
W	The array with 256 word variables
v	Variable number: index into a variable array
i	Word value
←	Assignment: copy a value to a variable
~	Inverse (negation)
∪	Boolean or operator (true if one of the arguments is true)

&	Boolean and operator (true if both arguments are true)
%	modulo operation: the remainder of a division

Examples:

Expression	*Meaning*
B[v]	The boolean value of the variable v.
W[v]	The word value of the variable v.
W[R % 256]	The word value of the word array indexed by the register.
R ← R + 1	Increment the register.

General instructions

Mnemonic	*Semantics*	*Description*
BLK	If (A) BLK ... END	Executes instructions inside the block only when accu is true. Jump to END if accu is false. The value of accu is preserved at the end of the block (stays true).
NBLK	If (~A) NBLK ... END	Complementary to BLK.
END		Marks the end of a block
STRT	If (A) start a game	Load a new game definition file identified by the register, found in the directory specified in the GAME variable, only when accu is true. Notice that the GAME variable is read after processing all instructions of the control cycle.
NSTRT	If (~A) start a game	Complementary to STRT.
TST	If (A) log word value v	Write the value of word value v to the status log, if accu is true.
NTST	If (~A) log word value v	Complementary to TST.
BRK	If (A) break emulator	If accu is true, stop running the emulator (so enter single step mode).
NBRK	If (~A) break emulator	Complementary to BRK.

Boolean instructions

Mnemonic	*Semantics*	*Description*
LDI i	A ← ~(i = 0)	If integer i equals 0, load accu with false. Otherwise load accu with true.
LD v	A ← B[v]	Load accu with boolean variable v.
NLD v	A ← ~B[v]	Load accu with inverse of boolean variable v.

Mnemonic	*Semantics*	*Description*
SET v	If (A) B[v] ← true	If accu is true, set boolean variable v to true.
NSET v	If (~A) B[v] ← true	If accu is false, set boolean variable v to true.
RST v	If (A) B[v] ← false	If accu is true, reset boolean variable v to false.
NRST v	If (~A) B[v] ← false	If accu is false, reset boolean variable v to false.
AND v	A ← A & B[v]	If variable v is false, assign false to the accu. Assign (accu and variable v) to accu.
NAND v	A ← A & ~B[v]	If variable v is true, assign false to the accu. Assign (accu and not variable v) to accu.
OR v	A ← A ∪ B[v]	If variable v is true, assign true to the accu. Assign (accu or variable v) to accu.
NOR v	A ← A ∪ ~B[v]	If variable v is false, assign true to the accu. Assign (accu or not variable v) to accu.
XOR v	A ← A & B[v]	If accu is unequal to variable v, assign true to accu, else false. Assign (accu or exclusive variable v) to accu.
NXOR v	A ← A & ~B[v]	If accu is equal to variable v, assign true to accu, else false. Assign (accu or exclusive not variable v) to accu.

Word instructions

Please note the following:

1. All instructions below are only executed if the accumulator is true. Otherwise these are skipped.
2. Also, all instructions are also available as the negated variant (NMVI, NMV, etc). These instructions have the same semantics, but are executed only if the accumulator is false.
3. All arithmetic is unsigned modulo 65536 (16-bit).

Mnemonic	*Semantics*	*Description*
MVI i	R ← i	Move the integer i into the register.
MV v	R ← W[v]	Move the word variable v into the register.
ST v	W[v] ← R	Store the register into word variable v.
MVX v	W[R % 256] ← W[v]	Move the word variable v into the word variable indexed by the register.
STX v	W[v] ← W[R % 256]	Store the word variable index by the register into the word variable v.
INCR	R ← R + 1	Increment the register.
DECR	R ← R - 1	Decrement the register.
INC v	W[v] ← W[v] + 1	Increment word variable v.

Mnemonic	*Semantics*	*Description*
DEC v	W[v] ← W[v] - 1	Decrement word variable v.
ADD v	R ← R + W[v]	Addition.
SUB v	R ← R - W[v]	Subtraction.
MUL v	R ← R * W[v] W[RESIDUE] ← R * W[v] / 65536	Multiplication where W[RESIDUE] holds the overflow[3].
DIV v	R ← R / W[v] W[RESIDUE] ← R % W[v]	Integer division where W[RESIDUE] holds the remainder[3].
MOD v	R ← R % W[v]	Modulo operation.
CE v	A ← R = W[v]	Compare equal
CNE v	A ← ~(R = W[v])	Compare not equal
GT v	A ← R > W[v]	Greater-true
GE v	A ← R >= W[v]	Greater or equal
LT v	A ← R < W[v]	Lower-true
LE v	A ← R <= W[v]	Lower or equal
RND	R ← 0 to (R - 1)	Take a random value from 0 upto (but not including) the register.

4.3 Example

An example of a hello-world game is (start a timer, start playing sample 1, wait until playing is finished, wait until the timer elapsed 10 seconds, then leave the game):

```
#include ../include/start.sct

LD RESET                          ; A ← B[RESET]
    ; Game is just started, place your initialization code here
    MVI 0                         ; If(A) R ← 0
    ST TIMER_1                    ; If(A) W[TIMER_1] ← R
    SET TIMCTRL_1                 ; If(A) Start timer 1 with 0.

NLD DONE
    MVI 0                         ; If(A) R ← 0
    CE STATE                      ; If(A) A ← (R = W[STATE])
          SET PLAY
          MVI 1                   ; If(A) R ← 1
          ST SND_FILE             ; If(A) W[SND_FILE] ← R
          MVI SOUND               ; If(A) R ← SOUND
          ST SND_TYPE             ; If(A) W[SND_TYPE] ← R
          INC STATE               ; If(A) Increment the current state
          SET DONE                ; If(A) Done for this control cycle

NLD DONE
```

[3]The value of RESIDUE is normally not preserved between cycles.

```
    MVI 1                       ; If(A) R ← 1
    CE STATE                    ; If(A) A ← (R = W[STATE])
    BLK                         ; If(A) Then:
          NLD PLAYING           ; A ← B[PLAYING]
                RST PLAY        ; If(A) Stop playing when done
                INC STATE       ; If(A) Increment the current state
                SET DONE        ; If(A) Done for this control cycle
    END

NLD DONE
    MVI 2                       ; If(A) R ← 2
    CE STATE                    ; If(A) A ← (R = W[STATE])
          MVI 100               ; If(A) R ← 100
          LT TIMER_1            ; If(A) A ← (R < W[TIMER_1])
                ; Timer_1 > 10 seconds
                MVI 60100       ; If(A) R ← 60100
                ST STATE        ; If(A) W[STATE] ← R
                SET DONE        ; If(A) Done for this control cycle

#include ../include/end_of_game.sct
#include ../include/end.sct
```

A more compact version of the same code is:

```
#include ../include/start.sct

LD RESET
    ; Game is just started, place your initialization code here.
    MVI 0  ST TIMER_1
    SET TIMCTRL_1

NLD DONE MVI 0 CE STATE
    SET PLAY
    MVI 1  ST SND_FILE
    MVI SOUND ST SND_TYPE
    INC STATE SET DONE

NLD DONE MVI 1 CE STATE
BLK
    NLD PLAYING
          RST PLAY
          INC STATE SET DONE
END

NLD DONE MVI 2 CE STATE
    MVI 100 LT TIMER_1
          ; Timer_1 > 10 seconds.
          MVI 60100 ST STATE SET DONE


#include ../include/end_of_game.sct
#include ../include/end.sct
```

Some remarks:

- We use the word variable STATE to hold the state we're currently in.

- As soon as we're done for this control cycle, set DONE. This ensures no instructions will be executed anymore. At the beginning of each cycle, DONE is reset in ../include/start.sct.
- When you set PLAY, Swinxs start playing and sets the boolean output variable PLAYING. As soon as the sample has been played, Swinxs will reset PLAYING.

5. SwinxsCoreTalk subsystems

5.1 File system

The Swinxs hooks up to a PC by simulating an USB-memory stick and appears to the PC user as a removable disk. Programming Swinxs basically involves nothing more than transferring files to the device. Other then that Swinxs assumes a file structure as shown below:

/boot.bin				*Boot game definition*
/00000.wav				*Sound file(s) for boot game*
/eng/				*English games and audio*
	sys/	00000.wav		*System sound file(s)*
	bracelets/	BL0.wav		*Names of XS*
		00008721.wav		*Personalized XS audio*
	numbers/	00000.wav		*Cardinal numbers*
		00001th.wav		*Ordinal numbers*
	games/	00000/	00000.bin	*Game selection game*
			00000.wav	*Game selection audio file(s)*
		00001/	00000.wav	*Jukebox songs*
		00011/	game.prp	*Game properties*
			00000.bin	*Game number 11*
			00000.wav	*Title of game*
			00001.wav	*Other audio file(s)*
		:		
		:		
/nld/				*Dutch games*
:				
:				

Filenames

Most filenames consist of 5 digits, followed by some optional text, a dot and the extension. The optional text is ignored. The 5 digits must be in the range of 0 to 65535 and correspond with word values used by the games.
The following extensions are used:

- WAV: uncompressed wave audio
- SCT: SwinxsCoreTalk files with game definitions
- BIN: compiled SwinxsCoreTalk files
- PRP: property files (plain VAR=VALUE text files)
- XML: XML files like scores

Personalized XS audio files always consist of 8 digits representing their 24-bit unique ID.

Root

In the root of the Swinxs memory-stick, you find the file boot.bin. This contains the boot game which runs when the Swinxs is powered on. This game can use language independent audio files located in the root.
Next you see a directory per supported language. This directory must be a three letter code, typically based on ISO 639-3.

Multiple languages

A language pack consists of:

- System audio files for generic use
- XS names
- Numbers and ordinals
- Game selection game
- Jukebox audio directory
- Several game directories

Games

Each game needs at least three files:

- game.prp which specify several game properties
- 00000.wav for the game title (used in the game selection menu)
- 00000.bin which contains the game definition

Furthermore, it can contain several game specific audio files and optionally several game definitions overlays (*.bin files).
Game definitions are binary objects produced by the SwinxsEmulator. The SwinxsEmulator compiles games written in SwinxsCoreTalk to binary code which can be executed by the Swinxs virtual machine.

A game is started by the STRT command:

```
MVI 48 ST GAME
MVI 4 STRT
```

The first two instructions store 48 in the game variable. Next the register is initialized with 4 and the game is started. What happens is that, after all other instructions of this cycle are executed, game 4 is loaded from the directory 48. So: /xxx/games/00048/00004.bin.
The next control cycle, the boolean variable RESET is set an the new game is executed.

5.2 RFID tags

We distinguish two kinds of RFID tags: XSes and accessories. With the former you can play a game. The latter can be used to influent game play, for example the Swinxs playing cards.
By default, RFID detection is turned off to save energy. So when you want to be able to detect any tag, first set the boolean output RF_ENA. And of course reset RF_ENA while you're not reading tags.

The first you must do to allow players, is to set the boolean output REGISTER (give it a rising edge). If REGISTER is high, and a new XS is detected, this XS is added to the player table as a new player. The table can hold upto 10 XSes. As soon as you lower REGISTER, no more XS can be added. The number of registered XSes is stored in the word variable NOF_XS.
When a tag is detected, the boolean output variable TAG gets high for a single control cycle. Furthermore, in RFID_LO, the RFID modulo 65530 is saved, together with RFID divided by 65530 which is stored in RFID_HI.
When a registered XS is detected, the boolean variable XS gets high for a single control cycle. The index of the player who's XS it is, is stored in the word variable LAST_XS. Registered XSes have two handy subsystems connected to: play the name of the XS and save the score per XS per game. So if you set WRITE_SCORE, all SCORE_* word variables for all registered XSes of the game are saved to disk.
When a accessory tag is detected, the boolean variable ACCESSORY get high for a single cycle.

5.3 Audio

Playing sound

Swinxs only supports uncompressed wave files. The virtual machine distinguishes the following audio types:

Type	*Directory*	*Purpose*
BOOT	/	Audio files only used by boot.bin. Should be language independent.
SYS	/xxx/sys	Audio files which can be used in any game and are hence game independent. See appendix for a list of all system samples.
NUMBER	/xxx/numbers	Audio files for the cardinal numbers 0 to 100 and 1000.
ORDINAL	/xxx/numbers	Audio files for the ordinal numbers 'first' to 'twentieth'.
BRACELET	/xxx/bracelets	Audio files to call XS colors. Also personalized XS names (based on RFID) are stored here.
SOUND	/xxx/games/xxxxx	Game specific audio files.

To start playing an audio sample, issue the following commands:

```
MVI SYS ST SND_TYPE
MVI 12 ST SND_FILE
SET PLAY
```

This will play audio file 12 from the sys-directory in the next control cycle. Also the boolean input PLAYING is set. When playing is done, PLAYING will be reset.
You should keep the boolean output PLAY high as long audio is played. To abort playing audio, just reset PLAY.
By default, the REPEAT boolean input variable is set when a game starts. This has the effect that when a sample is finished (and the PLAYING input is reset), the sample will automatically be played again. As a side effect, you can chain different samples very easily, without lowering PLAY in between:

```
NLD DONE MVI 0 CE STATE NAND PLAYING
    SET PLAY                ; Play
    MVI 9 ST SND_FILE ; sample 9
    MVI SYS ST SND_TYPE     ; from the system directory.
    INC STATE SET DONE      ; Increment the current state.

NLD DONE MVI 1 CE STATE NAND PLAYING
    SET PLAY                ; Play
    MVI 777 ST SND_FILE     ; sample 777
    MVI SOUND ST SND_TYPE   ; from the games directory.
    INC STATE SET DONE      ; Increment the current state.

NLD DONE MVI 2 CE STATE NAND PLAYING
    RST PLAY                ; Silence the Swinxs
```

To call an XS, use the sound type BRACELET. Swinxs will automatically handle name conflicts. As soon we have more then you player with a red XS, the first XS registered will be called 'red'. But the second will be called something special like “Red Nose”.
Also, if Swinxs calls an XS, it will first try to find a personalized XS-name: an audio file with the rfid as filename.
From Swinxs firmware 310, we have also background sounds available which Swinxs will mix real time with foreground sounds. This works only when both samples have the sample sample rate. Also, only SND_TYPE_BG of type SOUND, SYS, and BOOT are available. Background sounds are repeated until you turn them of by lowering PLAY_BG.

Voice recordings

From firmware version 100 or higher, Swinxs can make voice recordings. Audio is sampled at 8 kHz mono. Also Swinxs can personalize XS-tags by making a voice recording of a player's own name and attach the recording automatically to the XS.
To start a voice recording for tag <xs> (where <xs> is a number between 0 to the number of registered players), use the following code snippet:

```
MVI <xs> ST SND_FILE
MVI BRACELET ST SND_TYPE
SET RECORD
```

When you're done recording, reset the RECORD boolean. From now on, the name of the particular XS will be the sample just recorded. Each time Swinxs calls this player, the recording will be used. To use the original sample again, the player should dock his/her Swinxs and remove the personalized tag via the site.
For a voice recording into the game directory, use something like:

```
MVI <id> ST SND_FILE
MVI SOUND ST SND_TYPE
SET OVERWRITE SET RECORD
```

when you reset RECORD, you'll get the sample <id>REC.WAV in the game directory. If OVERWRITE is reset, Swinxs will find the next unused id from <id> or up.

5.4 Move & shake & noise

You have a special boolean input variable NOISE which will get high as soon Swinxs detects some loud noise (yelling, clapping your hands, etc).
Furthermore Swinxs can detect two kinds of movements: a short move and a long shake. All triggers can be adjusted in your game via five word variables:

MOVE_TH	A threshold how sensitive the accelerator is. Higher values means less sensitive. Defaults to 10.
MOVE_CNT	A count how many moves must be detected before the output is set. Defaults to 4.
SHAKE_TH	A threshold how sensitive the accelerator is. Higher values means less sensitive. Defaults to 15.
SHAKE_CNT	A count how many moves must be detected before the output is set. Defaults to 8.
NOISE_TH	A threshold how sensitive the noise (speech) detection is. Higher values means less sensitive. Defaults to 120.

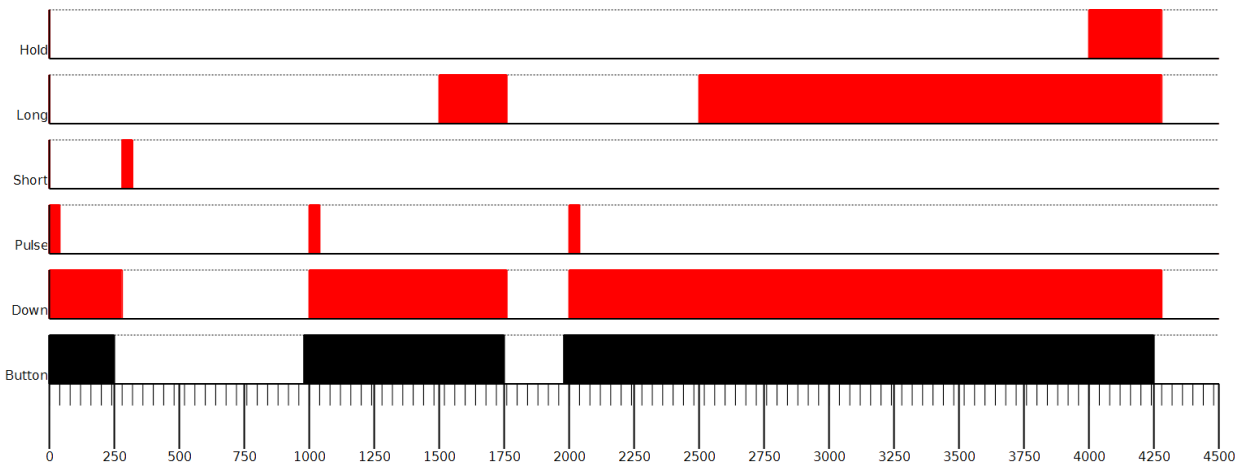
The defaults are set during the start of each game.

5.5 Buttons

For each three buttons we maintain four cases:

RED_DOWN	Set is soon a the button is pressed.
RED_SHORT	High for exactly one control cycle, if the button is pressed shortly (released with 500 msec).
RED_LONG	As soon the the button is pressed and hold down for more than 500 msec, this output is set high.
RED_HOLD	As soon the the button is pressed and hold down for more than 2 seconds, this output is set high.

The following chronogram visualizes the timing of various events (times are in milliseconds):



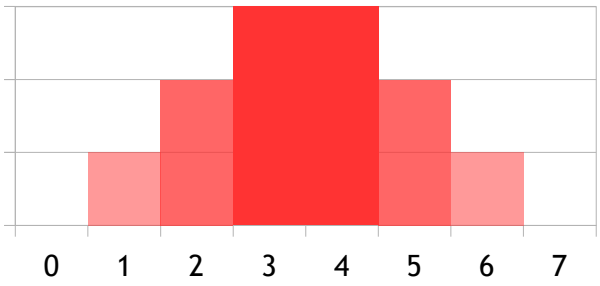
For GREEN we also have a special boolean variable 'GREEN_PULSE' to indicate that the green button is pressed (regardless of the duration). GREEN_PULSE will only be a single control cycle high. This variable is convenient when you don't use GREEN_LONG or GREEN_HOLD.
Note that some buttons already have a special meaning and are reserved:

WHITE_HOLD	Turn Swinxs off.
WHITE_SHORT	Alter Swinxs volume.
RED_LONG	Start the game selection menu.

Note that in general you should use the green button for the 'happy flow' and red for the 'unhappy flow'.

5.6 Pulse

Using the pulse subsystem you make the LEDs pulse in a specific color. A pulse consists of 6 steps (phases) and uses three different colors (intensities):



A pulse takes roughly 250 msec times the value you store in PULSE_PACE.
To start a pulse, do the following (red pulse as an example):

```
MVI RED_LO ST PULSE_COLOR_0
MVI RED_ME ST PULSE_COLOR_1
MVI RED_HI ST PULSE_COLOR_2
MVI 2 ST PULSE_PACE ST PULSE_TICK
INC PULSE_PHASE
```

To abort a running pulse, just use the preprocessor define:

```
STOP_PULSE
```

This will set the PULSE_PHASE to 0 and black out the LEDs.
The following colors can be used:

BLUE_LO, BLUE_ME, BLUE_HI	WHITE_LO, WHITE_ME, WHITE_HI
GREEN_LO, GREEN_ME, GREEN_HI	PINK_LO, PINK_ME, PINK_HI
RED_LO, RED_ME, RED_HI	ORANGE_LO, ORANGE_ME, ORANGE_HI
LEMON_LO, LEMON_ME, LEMON_HI	YELLOW_LO, YELLOW_ME, YELLOW_HI
CYAN_LO, CYAN_ME, CYAN_HI	BLACK_LO, BLACK_ME, BLACK_HI

VIOLET_LO, VIOLET_ME, VIOLET_HI GREY_LO, GREY_ME, GREY_HI
BLACK

The colors BLACK_* and GREY_* are special cases: we use red, green and blue for the tree black and cyan, lemon and violet for the three grey colors. BLACK is real black.
The following pulses are already predefined:

- Pulse lemon slowly while speaking (suppress this pulse by resetting PLAY_VISUAL)
- Pulse green, red or white quickly when pressing the corresponding buttons shortly. When you hold down the button, the color stays fixed.
- Pulse the color of XS when it is detected.
- Pulse red/green/blue when a non-registered tag is detected.
- Pulse cyan slowly while attached to USB (this is implemented in the Swinxs firmware, not the SwinxsCoreTalk applications).
- When an unhandled exception occurs, you will see a short red pulse.

5.7 Menu

Swinxs VM knows of three different menu types:

1. Language selection
2. Games selection
3. Sound file selection

A menu is basically a list through which we can iterate. Menu provides means to make a selection in your Swinxs application.
Menus can ordered alphanumerical or shuffled (randomized). The menu subsystem is initialized at the rising edge of MENU_ENA. Keep MENU_ENA set as long as you need to issue menu commands. To read the next element, set the boolean variable MENU_NEXT. Swinxs will reset MENU_NEXT for you at the end of the control cycle.
Because of limited shared Swinxs memory, the length of the shuffle list *plus* the number of menu items, must be below 180. This memory is freed only when you reinitialize the menu.

Language selection

Language selection is only meaningful in boot.bin. Once the user selects a language, and you start a game, you cannot switch a language anymore. Initialize MENU_TYPE with either M_LANGUAGE_SORT, or M_LANGUAGE_SHUF:

```
MVI M_LANGUAGE_SORT ST MENU_TYPE
SET MENU_ENA
```

The get the next language, just set the boolean output MENU_NEXT in the next control cycle. You can play all system audio files of a languages, e.g. /xxx/sys/*.wav.
Once you know which language the users wants, set the boolean output MENU_ACCEPT. Swinxs will then save the language to disk.
You know whether a user selected a language by testing the boolean input variable HAS_LANG.

Game selection

Selecting a game is similar. Initialize MENU_TYPE with either M_GAME_SORT, or M_GAME_SHUF:

```
MVI M_GAME_SHUF ST MENU_TYPE
SET MENU_ENA
```

When you set MENU_READ, in the next control cycle the next game is selected. This will fill the following word variables for you:

GAMES	The game id
MENU_SIZE	Number of games found
MINPLAY	Minimum number of players needed
MAXPLAY	Maximum number of players allowed
GAMETYPE	Game type
AGECLASS	Age class, to be defined

A typical game.prp files looks like this:

```
MINPLAY=1
MAXPLAY=10
GAMETYPE=1
AGECLASS=0
ID=<unique id>
TITLE=<the name of the game>
AUTHOR=<your name>
VERSION=1.0
LANGPACKID=4
```

TITLE is the title of the game, used at the Swinxs.com site. AUTHOR is your name.
The field ID should contain a unique ID to distinguish your game form others. To get this ID, please go to the website www.swinxs.com/register_game to register your game and obtain this ID.
When you record your own samples in English, you should use 4 as the LANGPACKID. For Dutch, use 3. For other languages, please contact Swinxs.com.
VERSION is the game version to determine if the Swinxs owner needs to upgrade. The format is <xxx>.<yyy>, where <xxx> is the major version number and <yyy> the minor, both a three digit integer. When the major number of VERSION (1 in the example above) is zero, then the game will not be published if you do a production release. So when you develop a game, VERSION should be something like 0.1.
When a GAMETYPE modulo 10 equals 9, it is *not* offered in the game selection menu. Such a game might be selected using an accessory tag. Example of these kind of games are games that need special RFID accessories.
When GAMETYPE modulo 10 equals 1, Swinxs tries not to offer this game the first time you start Swinxs. Preferably, you don't want to start with the Jukebox.
Because playing audio of type SOUND uses the GAME variable to determine the game directory, you can play the game title while iterating over the list of games.
Warning: when you issue a MENU_READ, playback will be aborted. So make sure Swinxs isn't playing audio while iterating through all available games.

Audio selection

Selecting a audio file is similar. Initialize MENU_TYPE with either M_SOUND_SORT, or M_SOUND_SHUF:

```
MVI M_SOUND_SHUF ST MENU_TYPE
SET MENU_ENA
```

When you set MENU_READ, in the next control cycle the next audio file is selected. This will fill the following word variables for you:

SND_FILE	The audio file id
MENU_SIZE	Number of games found

Just issue:

```
MVI SOUND ST SND_TYPE
PLAY
```

and you song is played.

5.8 Timers

Swinxs has 16 timers of which you can use 13 freely in your applications: TIMER_0 up to TIMER_12. Every timer has a timer-control linked to it: TIMCTRL_0 up to TIMCTRL_12.
If a timer-control is set, then the timer is running: every 100 milliseconds, the timer will be incremented by one. As long as the control is reset, the timer is paused.
During the situation that TIMER_* equals zero, the timer-control TIMCTRL_* will be low. You can use this fact to check whether a timer has looped from 65355 to 0.

5.9 Shuffle

The Shuffle subsystem is an extension of the RND (random) operator. Use shuffle when you need something like shuffling a set of players.
To initialize shuffle, store the number of items to be shuffled in the word variable SHUFFLE. Then generate a rising edge on the boolean output variable SHUF_ENA by setting SHUF_ENA. Then each time you set SHUF_READ, you will get the next value of the shuffled list in the word variable SHUFFLE. This will be a number between 0 and the shuffle value you initialized the subsystem with.
If you reset SHUF_ENA, the subsystem will be disabled. When set, Swinxs will reset SHUF_READ for you at the end of the control cycle.
When you read the last item of the shuffled list, the next read will return you the first element again. In that case, the boolean output variable SHUF_DONE will go high. To test whether you have read all items of the shuffled list, you can test SHUF_DONE.
You can rewind the list by setting SHUF_REWIND. And combine it with SET SHUF_READ to get the first item of the shuffled list again.
Example to count from 0 to 9 in random order:

```
LD RESET
```

```
    MVI 10 ST SHUFFLE ; I want a list of 10 shuffled numbers.
    SET SHUF_ENA            ; Enable the shuffle subsystem.
    SET SHUF_READ           ; Read the first item immediately.

NLD DONE MVI 0 CE STATE NAND PLAYING
BLK
    SET PLAY                        ; Play
    MVI NUMBER ST SND_TYPE  ; Number
    MV SHUFFLE ST SND_FILE  ; 0 to 9.
    SET SHUF_READ                   ; Read the next item.
    LD SHUF_DONE
          RST PLAY                  ; Stop calling numbers.
          RST SHUF_ENA              ; Disable the shuffle subsystem.
          INC STATE                 ; Leave this state.
END
```

Because of limited shared Swinxs memory, the length of the shuffle list *plus* the number of menu items, must be below 180. This memory is freed only when you reinitialize the shuffle.

5.10 Game exit

Save scores

To save all scores to disk, just set WRITE_SCORES. For every registered XS, the entries in the SCORE_0 to SCORE_9 will be appended to /xxx/scores.xml. A record contains the XS rfid, the game id and the score.
Be careful that Swinxs is not playing any audio, for this will be aborted than.

Speak out

If you include “scores.sct”, you get some code to let Swinxs speak out the scores. Essentially, when you jump to state FINISH_SCORE, Swinxs will say something like “The winner is/are blue, green and red (all) with X point”, followed by the names of the other players and their score. Also, the scores are written to disk and the game is exited via state FINISH_INTRO, see then next paragraph.

End of game

If you include “end_of_game.sct”, you have the possibility to end you're game in a standard way. You get three extra states: FINISH_INTRO and FINISH_BUTTON. When you switch to that state, the followings happens:

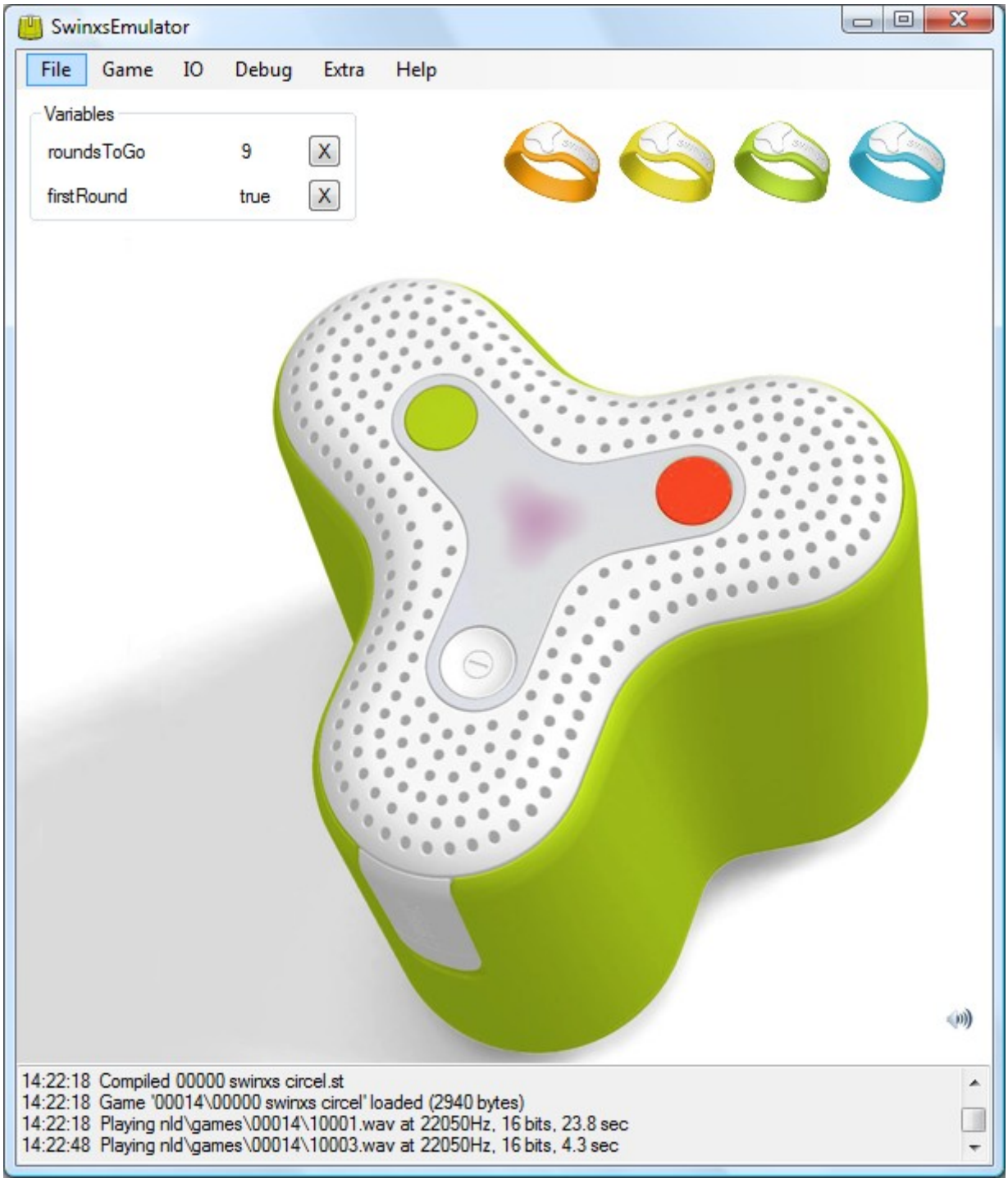
State	Actions
FINISH_INTRO	"Excellent work Swinxers. The game has officially ended." Jump to FINISH_BUTTON
FINISH_BUTTON	"Press the green button to play <game> again. Press the red button if you want to play another game." After time out of 8 seconds: “<game> has ended” Green button: exit game and start the game selection game (game 0)

5.11 Power management

To turn off Swinxs, set the boolean output variable SUICIDE. When Swinxs detects a low battery situation, it will set the boolean input BAT_LOW. The SwinxsCoreTalk game libraries will automatically handle this: at the first possible moment, Swinxs will say something like “please hook me up to a charger”. Also, the red LED will light softly if Swinxs doesn't use it's LEDs for something else.

6. SwinxsEmulator

One of the purposes of the emulator is to compile and run SwinxsTalk and SwinxsCoreTalk games.
You're referred to the SwinxsSDK documentation for more information concerning the SwinxsEmulator.



6.1 SwinxsCoreTalk Preprocessor

The emulator has a build in preprocessor for conditional compiling and for maintaining a library of code to be included in your applications. For this, you use compiler directives.

Includes

You can include other files using the include directive:

```
#include volume.sct
```

Defines

You can define a constant using:

```
#define DEBUG
#define I_AM_RUNNING BOOLVAR_0
```

When compiling a *.sct file inside a language directory, the predefined constant LANG_xxx can be used, where xxx is the language name. For example: LANG_ENG or LANG_NLD.
Another usage of defines is to give code more convenient names. When you define a constant X as Y, the preprocessor will substitute all occurrences of X with Y.
Defines are case sensitive and match whole words only.

Conditional compilation

The Swinxs compiler has conditional compilation directives:

```
#ifdef XXX
...
#else
...
#endif
```

and:

```
#ifndef XXX
...
#endif
```

Comments

Comments start with a semi colon. All text after a semi colon is ignored.

6.2 SwinxsCoreTalk Debugger

Typically, you Load boot.sct from the File menu. Swinxs is booted.

By pressing the F10 key, you start the debugger to single step your code. Press F5 to start running again.
Reboot Swinxs by pressing the Shift-Ctrl-F5 key.

Breakpoints

Sometimes it is handy to define your own breakpoints in your code. For this, add BRK or NBRK instructions. When the emulator reaches a break, the control cycle is aborted and you can single step your code.

Logging

A TST <word-var> will display the value of <word-var> in the status log if the accu is true.

6.3 Tips

Some arbitrary useful tips when developing your games:

1. To reload your current game, press shift-F5 followed by F5.
2. Don't forget to add a instruction like 'SET RF_ENA' if you want to detect tags.
3. The RESET part of your code is only executed the first control cycle. The first state is 0, which will normally be executed at the second control cycle.
4. Hence, but MENU and SHUFFLE initialization not in the RESET part. You need RESET to make sure the boolean input variables get low. Swinxs needs a rising edge to start initialization.